COMP 308
ARTIFICIAL INTELLIGENCE
PART 3.2 – UNINFORMED (BLIND) SEARCH

Njeri Ireri

Jan – April 2020

We Shall Discuss

- What is Uninformed Search?
- Uninformed Search Methods
  - Depth-first search
  - Breadth-first search
  - Non-deterministic search
  - Iterative deepening search
  - Bi-directional search

Uninformed Search?

- Simply searches the state space (or NET)
- Can only distinguish between goal state and non-goal state
- Sometimes called Blind search as it has no information or knowledge about its domain

Uninformed Search Characteristics

- Blind Searches have no preference as to which state (node) that is expanded next
- The different types of blind searches are characterised by the order in which they expand the nodes
  - This can have a dramatic effect on how well the search performs when measured against the four criteria we defined in an earlier lecture
  - Search evaluation criteria - Completeness, Time Complexity, Space Complexity, Optimality (of given solution when there are several solutions to choose from)

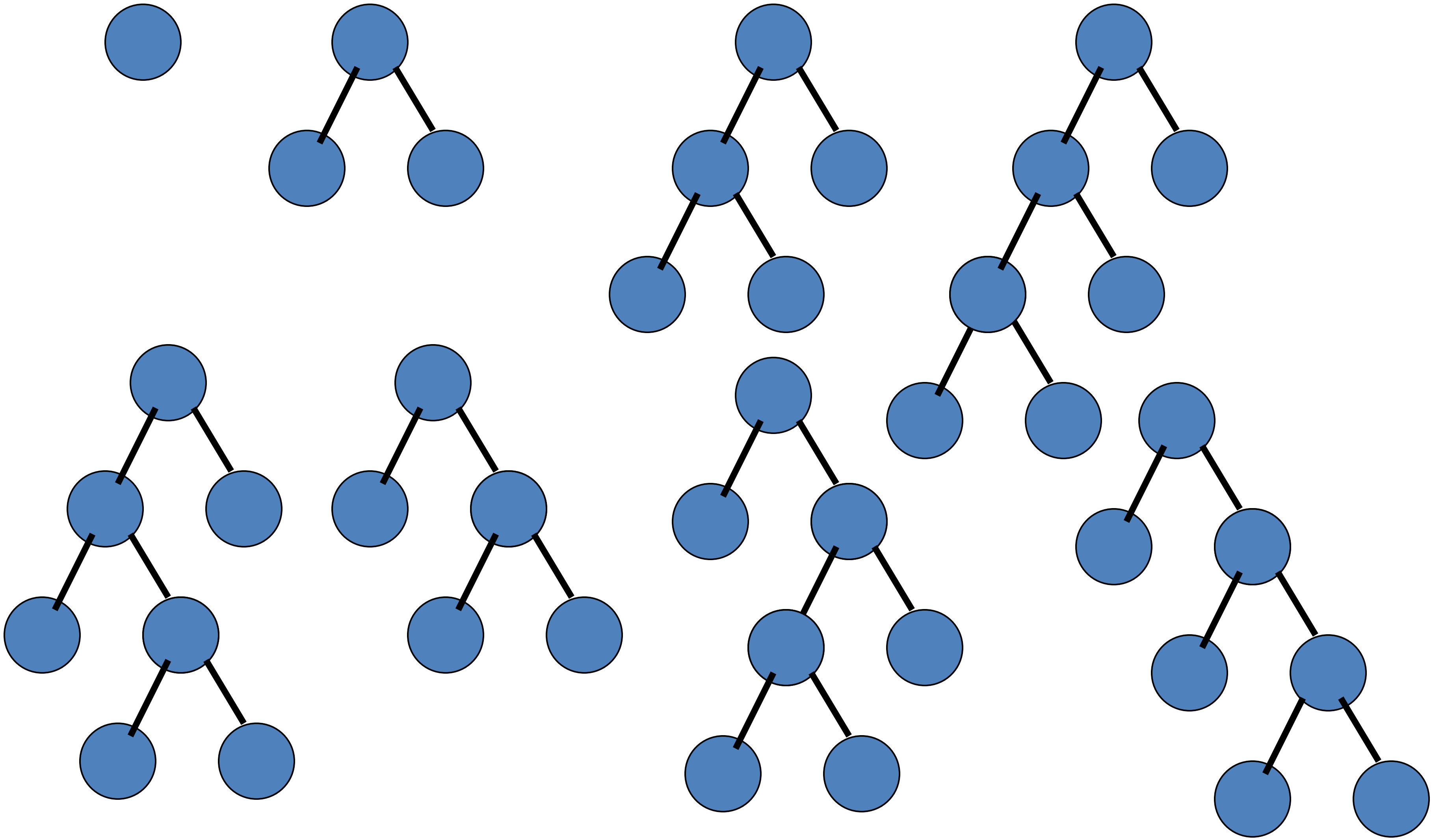
Uninformed (Blind) Search Methods

- Methods that do not use any specific knowledge about the problem
- These are:
  - Depth-first search
  - Breadth-first search
  - Non deterministic search
  - Iterative deepening search
  - Bi-directional search

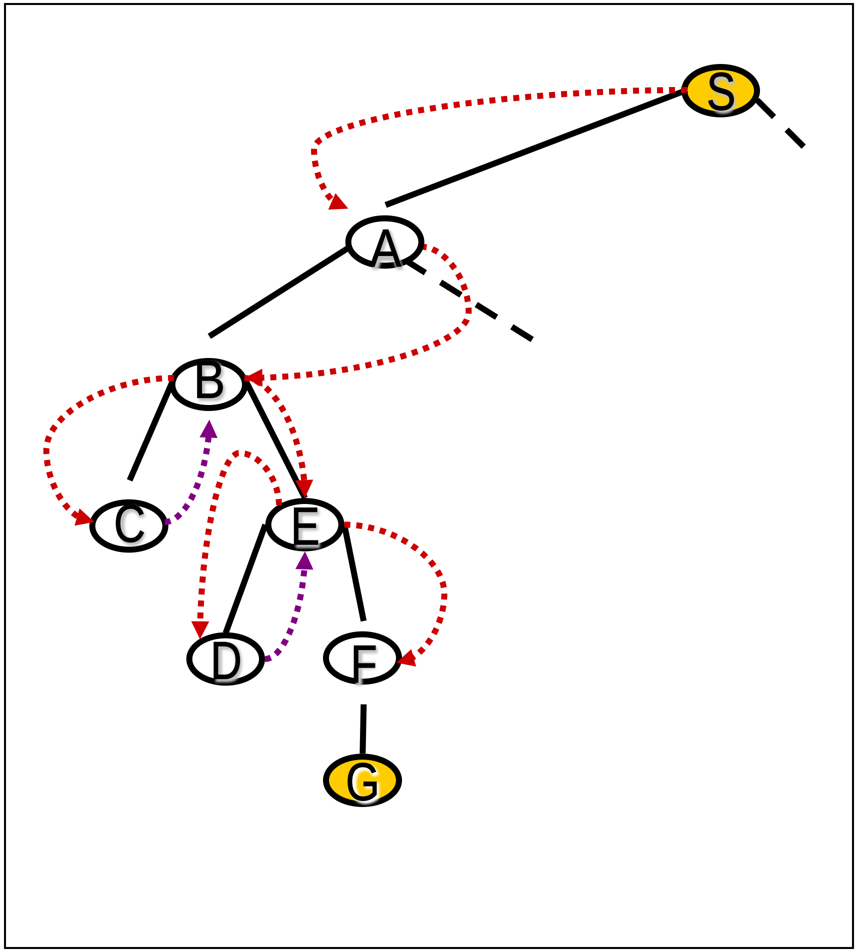
1. Depth-first Search

- Expand the tree as deep as possible, returning to upper levels when needed

Depth-First Search

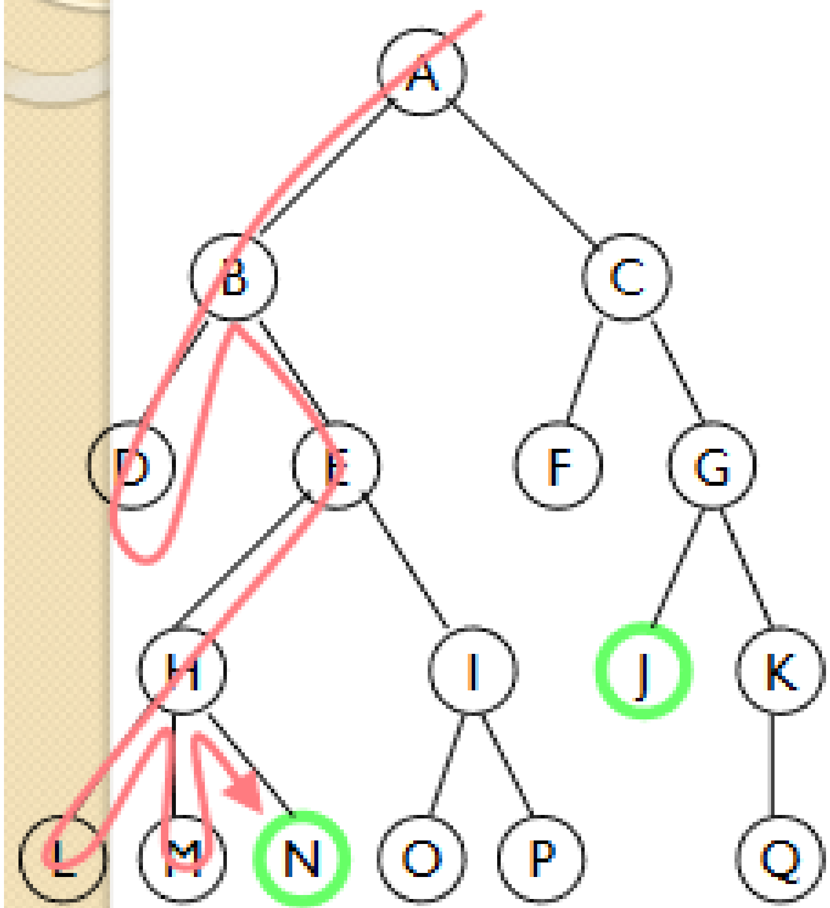


Depth-First Search
= Chronological backtracking



- Select a child
  - convention: left-to-right
- Repeatedly go to next child, as long as possible
- Return to left-over alternatives (higher-up) only when needed

Depth-First Search = Chronological backtracking



- A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order A B D E H L M N I O P C F G J K Q
- N will be found before J

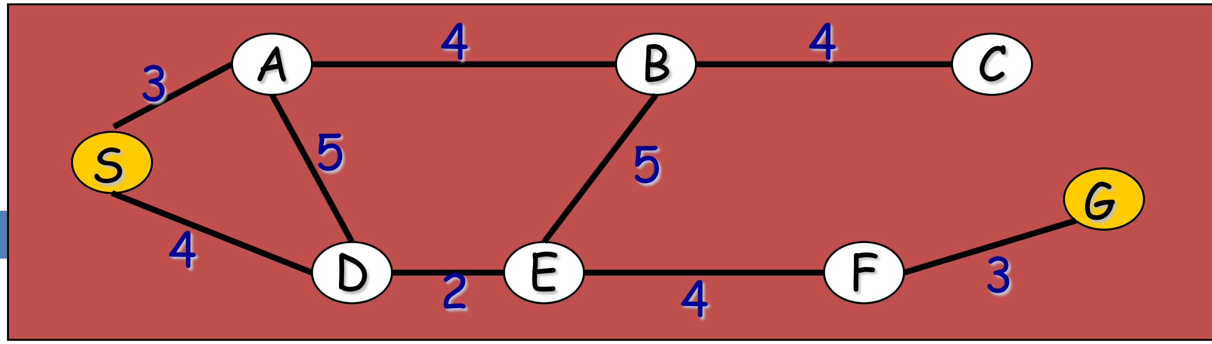
Depth-First Algorithm:

1. QUEUE <-- path only containing the root;

2. WHILE QUEUE is not empty
   AND goal is not reached

   DO remove the first path from the QUEUE;
   create new paths (to all children);
   reject the new paths with loops;
   add the new paths to front of QUEUE;

3. IF goal reached
   THEN success;
   ELSE failure;



1. QUEUE <-- path only containing the root;

2. WHILE QUEUE is not empty
   AND goal is not reached

   DO
   - remove the first path from the QUEUE;
   - create new paths (to all children);
   - reject the new paths with loops;
   - add the new paths to front of QUEUE;

3. IF goal reached
   THEN success;
   ELSE failure;

Trace of Depth-First for running example:

(S)	S removed, (SA,SD) computed and added
(SA, SD)	SA removed, (SAB,SAD,SAS) computed, SAB,SAD) added
(SAB,SAD,SD)	SAB removed, (SABA,SABC,SABE) computed, (SABC,SABE) added
(SABC,SABE,SAD,SD)	SABC removed, (SABCB) computed, nothing added
(SABE,SAD,SD)	SABE removed, (SABEB,SABED,SABEF) computed, (SABED,SABEF)added
(SABED,SABEF,SAD,SD)	SABED removed, (SABEDS,SABEDA.SABEDE) computed, nothing added
(SABEF,SAD,SD)	SABEF removed, (SABEFE,SABEFG) computed, (SABEFG) added
(SABEFG,SAD,SD)	goal is reached: reports success

Evaluation Criteria:

- **Completeness**
  - Does the algorithm always find a path?
    - (for every state space such that a path exits)
- **Speed** (worst time complexity) :
  - What is the highest number of nodes that may need to be created?
- **Memory** (worst space complexity) :
  - What is the largest amount of nodes that may need to be stored?
- Expressed in terms of:
    - d = depth of the tree
    - b = (average) branching factor of the tree
    - m = depth of the shallowest solution

Note: approximations !!

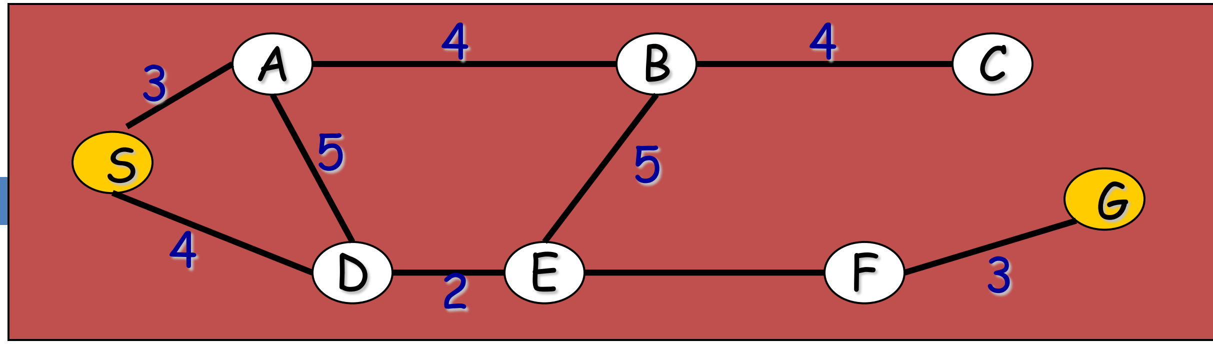
- In our complexity analysis, we do not take the built-in **loop-detection** into account
- The results only 'formally' apply to the variants of our algorithms **WITHOUT** loop-checks
- Studying the effect of the loop-checking on the complexity is hard:
  - The overhead of the checking MAY or MAY NOT be compensated by the reduction of the size of the tree
- <u>Also</u>: our analysis **DOES NOT** take the length (space) of representing paths into account !!

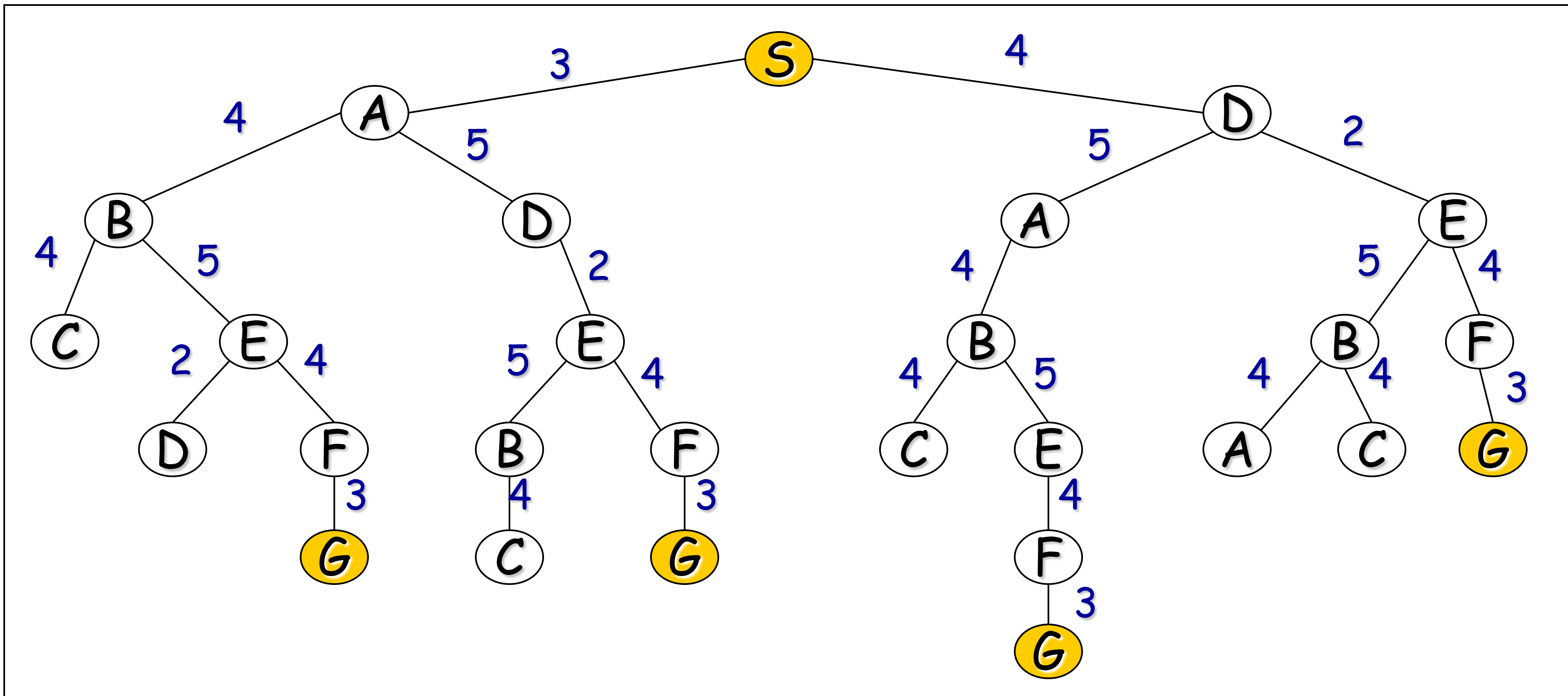
Completeness (depth-first)

- Complete for FINITE (implicit) NETS
  - (= State space with finitely many nodes)

- **IMPORTANT:**
  - This is due to integration of LOOP-checking in this version of Depth-First (and in all other algorithms that will follow) !
    - IF we do not remove paths with loops, then Depth-First is not complete (may get trapped in loops of a finite State space)

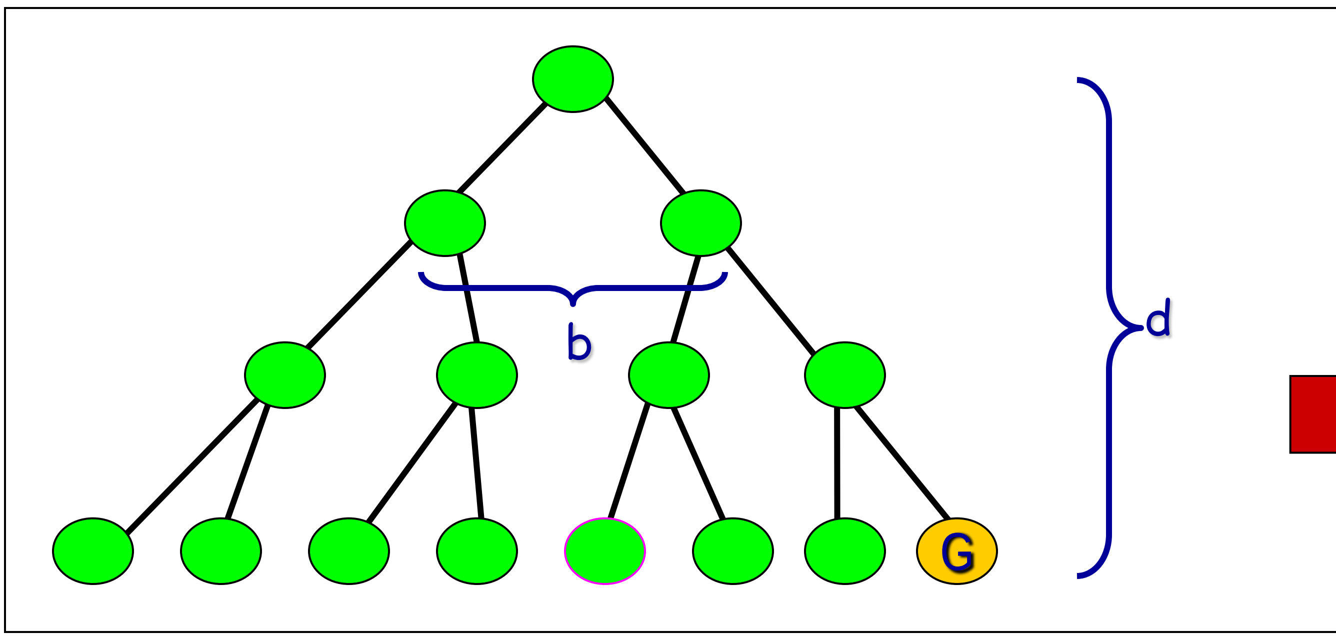
- **Note:** does NOT find the shortest path


S
A
B
C
D
E
F
G
3
4
4
5
5
4
2
3


S
A
D
B
C
E
F
G
3
4
4
5
5
2
4
2
3

Speed (depth-first)

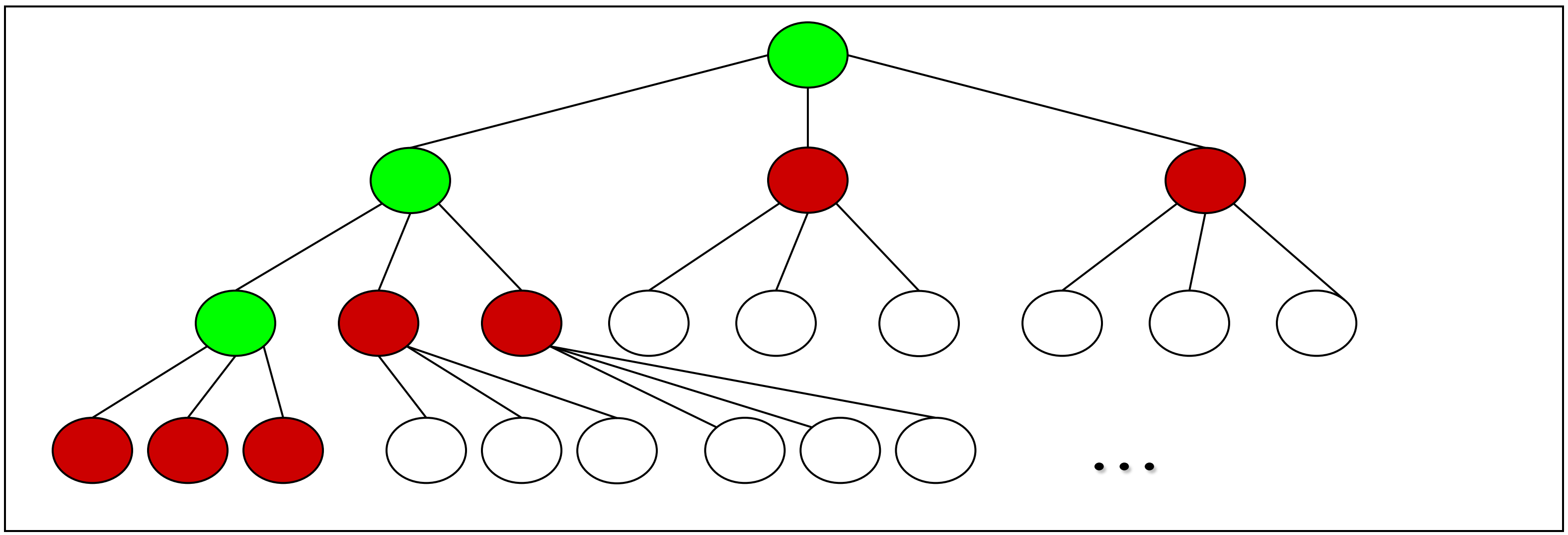
- In the worst case:
  - the (only) goal node may be on the right-most branch,

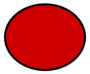


- Time complexity $== b^d + b^{d-1} + \ldots + 1 = \dfrac{b^{d+1} - 1}{b - 1}$
- Thus: $O(b^d)$

Memory (depth-first)

- Largest number of nodes in QUEUE is reached in bottom left-most node
- Example: d = 3, b = 3 :



  - QUEUE contains all  nodes. Thus: 7.
  - In General: ((b-1) * d) + 1
  - Order: O(d*b)

2. Breadth-First Search

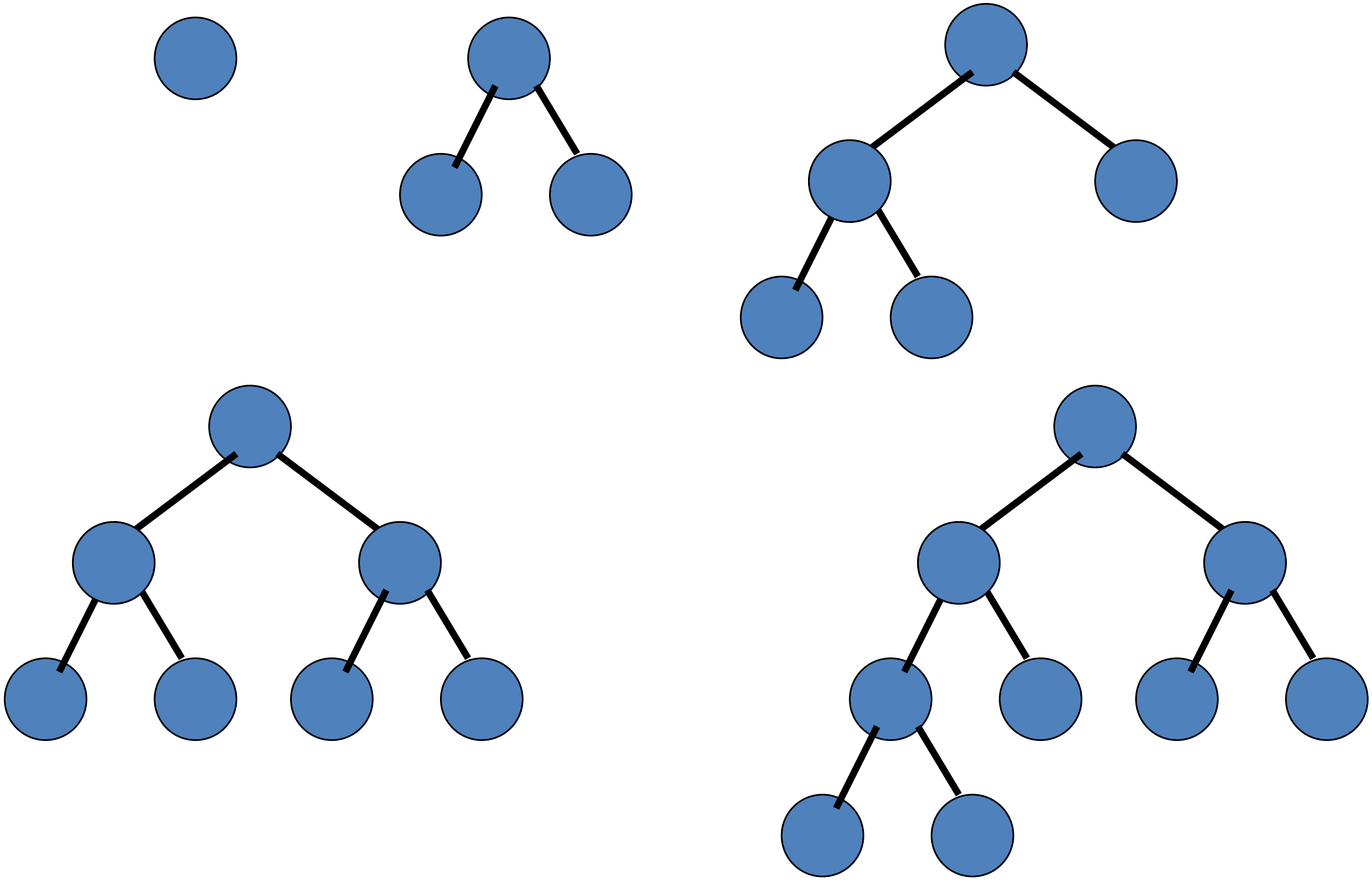
- Expand the tree layer by layer, progressing in depth.

- In other words,
  - Expand root node first
  - Expand all nodes at level 1 before expanding level 2

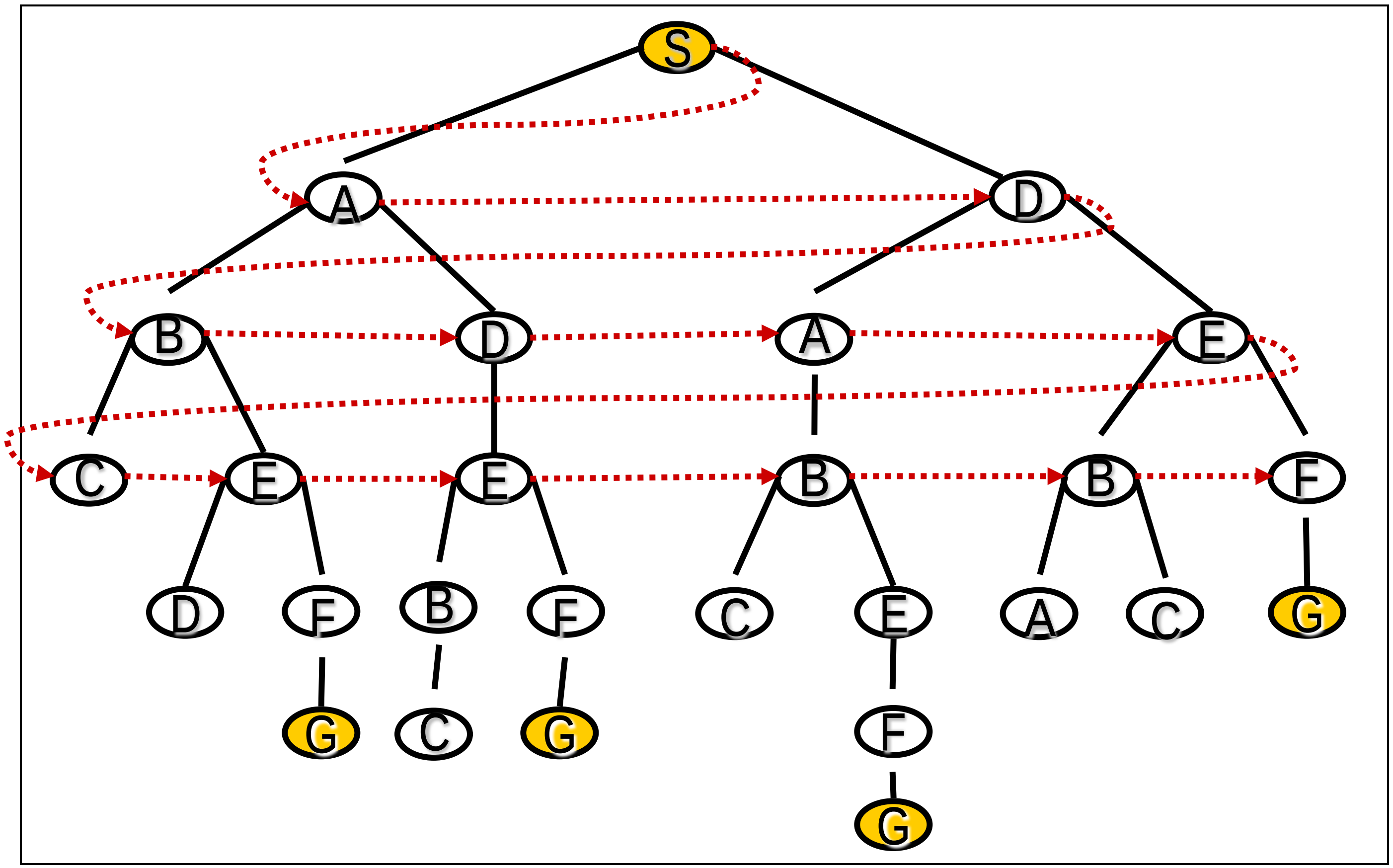
  OR

  - Expand all nodes at level d before expanding nodes at level d+1

Breadth-First Search

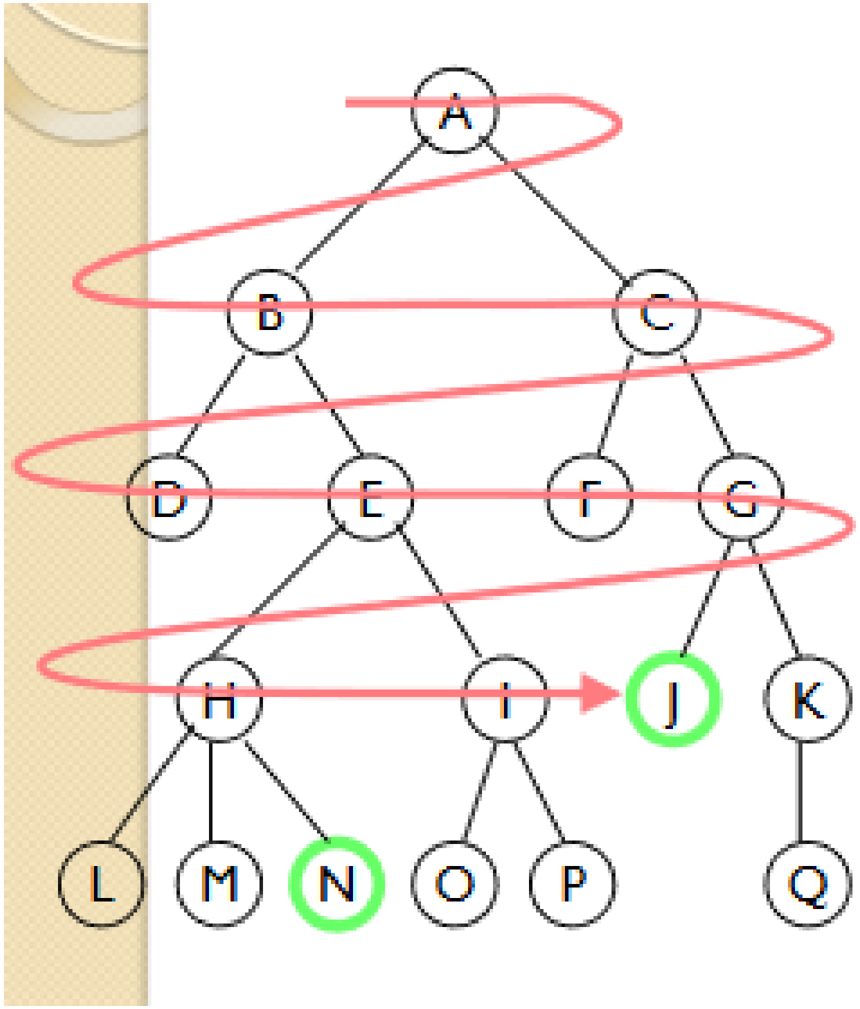


Breadth-First Search:



- Move downwards, level by level, until goal is reached

Breadth-First Search:



- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q
- J will be found before N

Breadth-First Algorithm:

1. QUEUE <-- path only containing the root;

2. WHILE QUEUE is not empty
AND goal is not reached

DO remove the first path from the QUEUE;
create new paths (to all children);
reject the new paths with loops;
add the new paths to back of QUEUE;

3. IF goal reached
THEN success;
ELSE failure;

ONLY DIFFERENCE !

Trace of breadth-first for running example:

- (S) — S removed, (SA,SD) computed and added
- (SA, SD) — SA removed, (SAB,SAD,SAS) computed, (SAB,SAD) added
- (SD,SAB,SAD) — SD removed, (SDA,SDE,SDS) computed, (SDA,SDE) added
- (SAB,SAD,SDA,SDE) — SAB removed, (SABA,SABE,SABC) computed, (SABE,SABC) added
- (SAD,SDA,SDE,SABC,SABE) — SAD removed, (SADS,SADA, SADE) computed, (SADE) added
- etc, until QUEUE contains:
- (SABED,SABEF,SADEB,SADEF,SDABC,SDABE,SDEBA,SDEBC, SDEFG)

  goal is reached: reports success

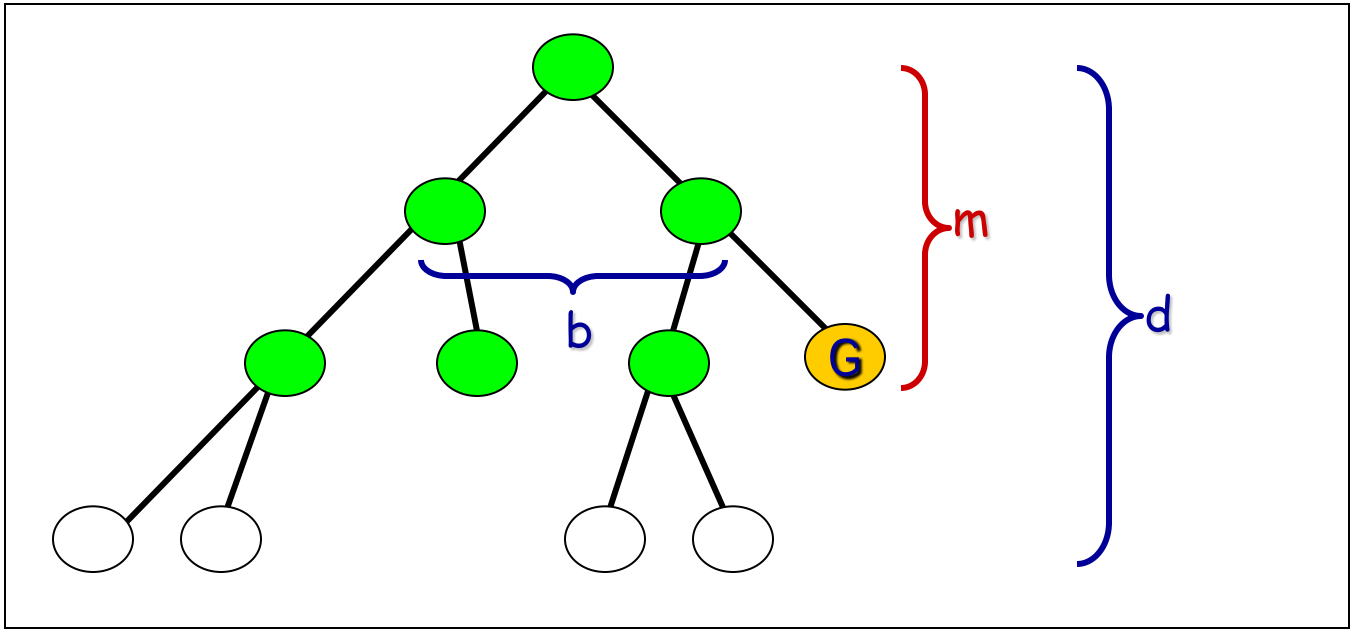
Completeness (breadth-first)

- **Complete**
  - even for infinite implicit NETS !
  - Would even remain complete without our loop-checking

- **Note:** ALWAYS finds the shortest path

Speed (breadth-first)

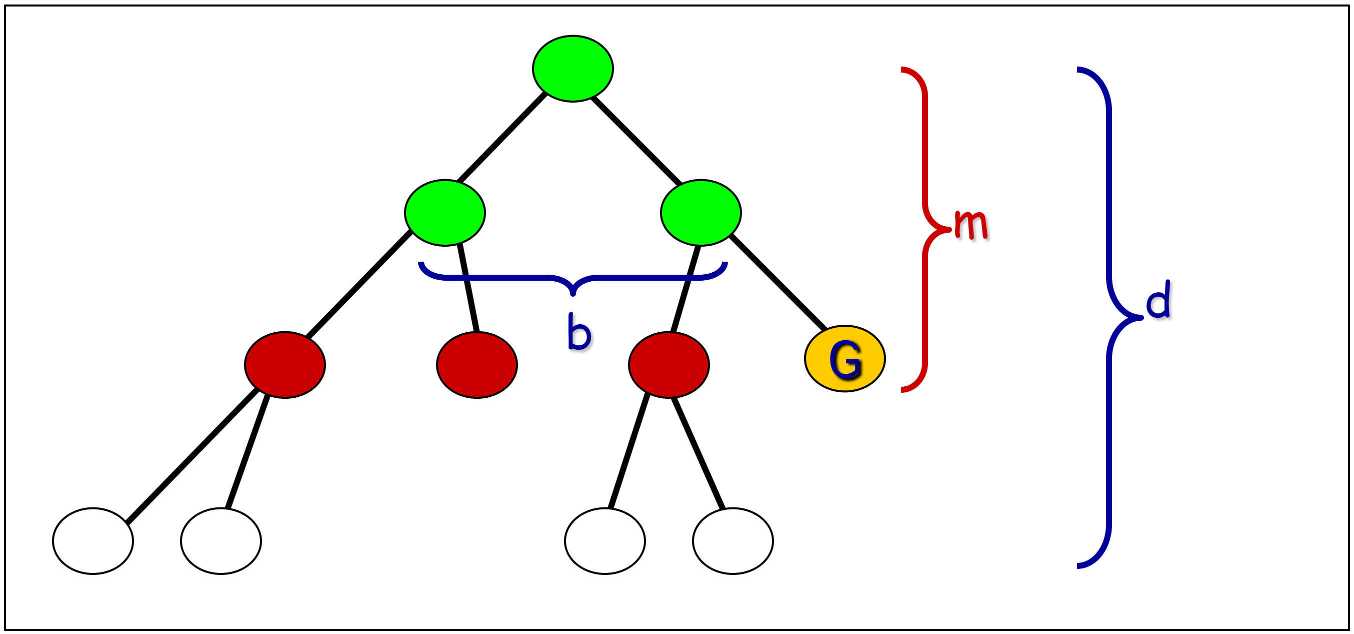
- If a goal node is found on depth m of the tree, all nodes up till that depth are created

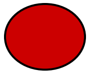


  - Thus: $O(b^m)$
  - **Note:** depth-first would also visit deeper nodes

Memory (breadth-first)

- Largest number of nodes in QUEUE is reached on the level m of the goal node



  - QUEUE contains all  and 

nodes. (Thus: 4)
  - In General: b^m
  - This usually is <u>MUCH</u> worse than depth-first !!

Exponential Growth (breadth-first)

Depth	Nodes	Time		Memory	
0	1	1	millisecond	100	kbytes
2	111	0.1	second	11	kilobytes
4	11,111	11	seconds	1	megabyte
6	10^6	18	minutes	111	megabytes
8	10^8	31	hours	11	gigabytes
10	10^{10}	128	days	1	terabyte
12	10^{12}	35	years	111	terabytes
14	10^{14}	3500	years	11,111	terabytes

- Time and memory requirements for breadth-first search, assuming a branching factor of 10, 100 bytes per node and searching 1000 nodes/second

Exponential Growth - Breadth-First Observations

- Space is more of a factor to breadth first search than time
- Time is still an issue. Who has 35 years to wait for an answer to a level 12 problem (or even 128 days to a level 10 problem)
- It could be argued that as technology gets faster then exponential growth will not be a problem. But even if technology is 100 times faster we would still have to wait 35 years for a level 14 problem and what if we hit a level 15 problem!

Practical Evaluation:

- **1.Depth-first search:**
  - IF the search space contains very deep branches without solution, THEN Depth-first may waste much time in them
- **2. Breadth-first search:**
  - Is VERY demanding on memory !
- **Solutions ??**
  - Non-deterministic search
  - Iterative deepening

3. Non-deterministic Search

- A Non-deterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm
- There are several ways an algorithm may behave differently from run to run

Non-deterministic Search

1. QUEUE <-- path only containing the root;

2. WHILE { QUEUE is not empty
AND goal is not reached

DO { remove the first path from the QUEUE;
create new paths (to all children);
reject the new paths with loops;
add the new paths in random places in QUEUE;

3. IF goal reached
THEN success;
ELSE failure;

4. Iterative Deepening Search

- Also referred to as **Iterative Deepening Depth-First Search**
- Restrict a depth-first search to a fixed depth
  - a depth-limited version of depth-first search is run repeatedly with increasing depth limits until the goal is found
- If no path is found, increase the depth and restart the search

Depth-limited Search

1. DEPTH <-- <some natural number>
   QUEUE <-- path only containing the root;

2. WHILE
   QUEUE is not empty
   AND goal is not reached

   DO
   remove the first path from the QUEUE;
   IF path has length smaller than DEPTH
   create new paths (to all children);
   reject the new paths with loops;
   add the new paths to front of QUEUE;

3. IF goal reached
   THEN success;
   ELSE failure;

Iterative Deepening Algorithm:

1. DEPTH <-- 1

2. WHILE goal is not reached

   DO { perform Depth-limited search;
   increase DEPTH by 1;

Iterative Deepening: the best 'blind' search

- Complete: yes - even finds the shortest path (like breadth first)
- Memory: $b*m$ (combines advantages of depth- and breadth-first)
- Speed:
  - If the path is found for Depth = m, then how much time was wasted constructing the smaller trees??
- $b^{m-1} + b^{m-2} + \ldots + 1 = \frac{b^m - 1}{b - 1} = O(b^{m-1})$
- While the work spent at DEPTH = m itself is $O(b^m)$

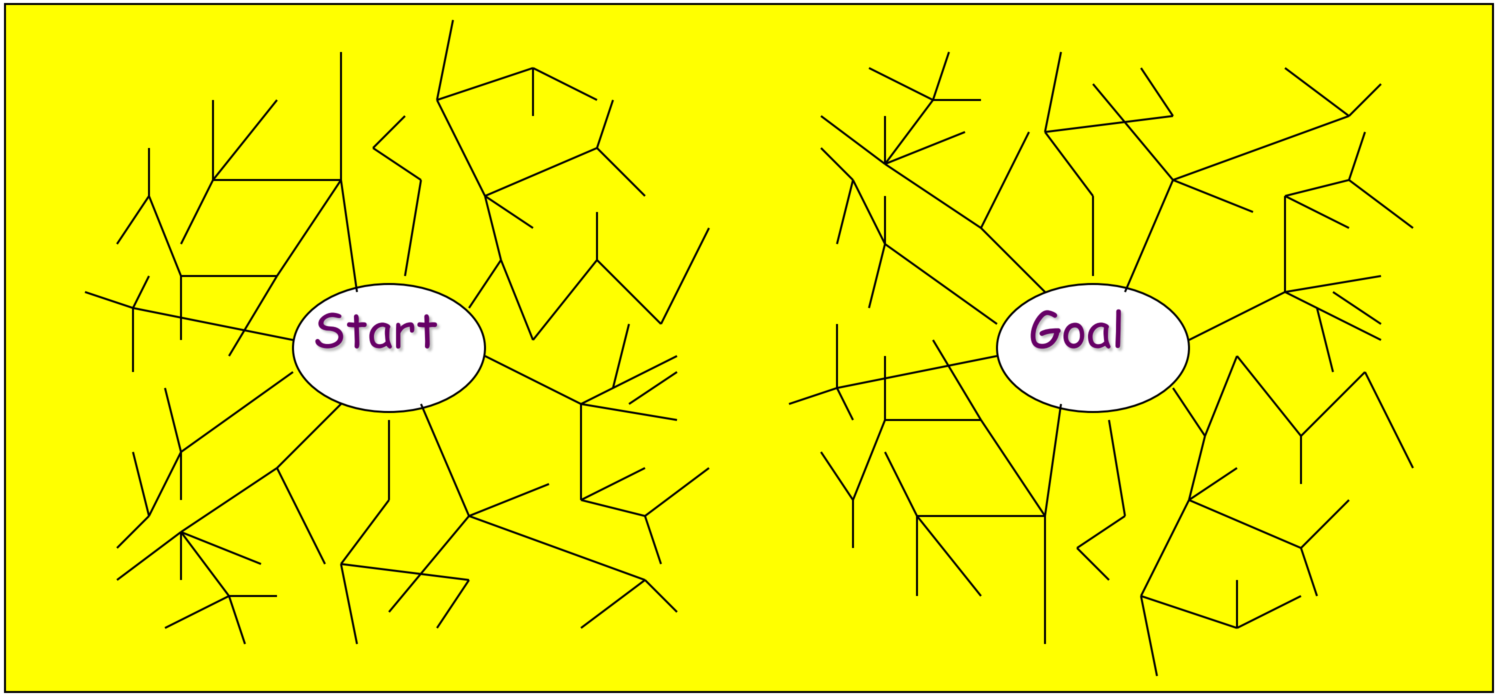
In general: VERY good trade-off

5. Bi-directional Search

- Compute the tree from the start node and from a goal node, until these meet

Bi-directional Search

- IF you are able to EXPLICITLY describe the GOAL state, AND you have BOTH rules for FORWARD reasoning AND BACKWARD reasoning:



Bi-directional Algorithm:

1. QUEUE1 <-- path only containing the root;
   QUEUE2 <-- path only containing the goal;

2. WHILE both QUEUEi are not empty
   AND QUEUE1 and QUEUE2 do NOT share a state

   DO { remove their first paths;
   create their new paths (to all children);
   reject their new paths with loops;
   add their new paths to back;

3. IF QUEUE1 and QUEUE2 share a state
   THEN success;
   ELSE failure;

Properties (Bi-directional):

- Complete: Yes.
- Speed: If the test on common state can be done in constant time (hashing):
  - $2 * O(b^{m/2}) = O(b^{m/2})$
- Memory: similarly: $O(b^{m/2})$

Exercise

- Confirm the trace of depth-first search algorithm given in slide 11
- Confirm the trace of breadth-first search algorithm given in slide 22
- Uniform Cost Search: find out how this search algorithm works

Uniform-Cost Search Algorithm

- If all the edges in the search graph do not have the same cost then breadth-first search generalizes to uniform-cost search. Instead of expanding nodes in order of their depth from the root, uniform-cost search expands nodes in order of their cost from the root. At each step, the next step n to be expanded is one whose cost g(n) is lowest where g(n) is the sum of the edge costs from the root to node n. The nodes are stored in a priority queue. This algorithm is also known as Dijkstra's single-source shortest algorithm.
- UCS is Dijkstra's algorithm which is focused on finding a single shortest path to a single finishing point rather than the shortest path to every point. UCS does this by stopping as soon as the finishing point is found.
- Whenever a node is chosen for expansion by uniform cost search, a lowest-cost path to that node has been found. The worst case time complexity of uniform-cost search is $O(b^c/m)$, where c is the cost of an optimal solution and m is the minimum edge cost. Unfortunately, it also suggests the same memory limitation as breadth-first search.

Uniform-Cost Search Algorithm

- with f(n) = the sum of edge costs from start to n

f(n) = "cost from **start** to **n**"